# Source Code Patches from Dynamic Analysis

Indigo Orton
indigo.orton@cl.cam.ac.uk
University of Cambridge
Department of Computer Science and Technology
Cambridge, UK

Alan Mycroft
alan.mycroft@cl.cam.ac.uk
University of Cambridge
Department of Computer Science and Technology
Cambridge, UK

## ABSTRACT

Dynamic analysis can identify improvements to programs that cannot feasibly be identified by static analysis; concurrency improvements are a motivating example. However, mapping these dynamic-analysis-based improvements back to patch-like source-code changes is non-trivial. We describe a system, *Scopda*, for generating source-code patches for improvements identified by execution-trace-based dynamic analysis. *Scopda* uses a graph-based static program representation (*abstract program graph, APG*), containing inter-procedural control flow and local data flow information, to analyse and transform static source-code. We demonstrate *Scopda*'s ability to generate sensible source code patches for Java programs, though it is fundamentally language agnostic.

## CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; Object oriented languages; Concurrent programming languages; **Software performance**; *Source code generation*; **Automated static analysis**; **Dynamic analysis**; *Translator writing systems and compiler generators*; • **Theory of computation** → *Concurrency*; **Program analysis**.

## KEYWORDS

Source patch generation, Dynamic-to-static resolution, Concurrency improvements, Task-based concurrency, Abstract program graph

**ACM Reference Format:**
Indigo Orton and Alan Mycroft. 2021. Source Code Patches from Dynamic Analysis. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '21), July 13, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3464971.3468416

## 1 INTRODUCTION

Dynamic analysis can identify improvements for programs that cannot be identified using static analysis. In particular, dynamic analysis is well suited to identifying improvements in performance and concurrency, and especially concurrency performance. However, implementing these improvements is non-trivial as the data they use and emit are not directly connected to the static program structure. This barrier to implementation reduces the utility and adoption of such dynamic analyses.

We describe a system, *Scopda*, for generating source-code patches to transform programs following dynamic analysis. It uses the *Rehype* [7] companion tool to identify *improvements* (instances of *optimisations*[1]) to a program's concurrency performance using execution-trace-based dynamic analysis (Section 2). *Rehype* generates *improvement specifications*, based on trace data, that can point a developer towards a source code *change* to be made, but leaves interpreting the specification to implement the *change* to the developer. This interpretation is non-trivial and open to errors.

*Scopda* (Section 3) automatically generates concrete source-code *patches* for the *improvements* identified by *Rehype*. A *patch* is the git-style diff between the source code pre- and post-change. *Scopda* maps the dynamic-domain *improvement specifications* into the static domain and then generates source-code *patches* to implement them. While there is a single, generalised method for dynamic-to-static mapping, *Scopda* implements a specialised *change transformation function* (CTF) for each optimisation.

An *improvement specification* consists of an *optimisation type*, a *caller-path*, and a *callee-tree*. The *optimisation type* determines the appropriate CTF, while the *caller-path* and *callee-tree* determine[2] the specification's *dynamic context*. The *dynamic context* approximates where the *improvement* should be made. *Caller-paths* and *callee-trees* are structured sets of function *invocations*. For a given invocation, the *caller-path* is the series of invocations that contain it, and the *callee-tree* contains the invocations it triggers.

To implement an *improvement*, the *dynamic context* must be mapped into a *static location*. A *static location* is a specific location within the source code (e.g. a function call). In the main example in this paper the mapping is one-to-one, however, in more complex situations a *dynamic context* may map to multiple *static locations* (see Section 3.2). *Scopda* explores execution paths in the program to map *dynamic contexts* to *static locations*. This execution is performed on an *abstract program graph* (APG), a static program representation that contains inter-procedural control flow and local data flow. After calculating the *static locations*, *Scopda* performs graph transformations on the APG to implement the *improvement* and renders the modified APG as source code.

Dynamic-to-static mapping is non-trivial as it has to match sparse dynamic traces (*caller-paths* and *callee-trees*) to concrete *static locations*. The dynamic traces are sparse as *Rehype* only traces a subset of the program's functions (the *tracked* functions) and does not trace sub-function information (e.g. branches). This sparse tracing is necessary to limit the tracing overhead. Too much overhead

[1] In this paper we only consider the *convert-to-inline* optimisation from *Rehype*.
[2] Intuitively the *dynamic context* is exactly the *caller-path* but since the *caller-path* traces entry points, rather than call sites, information from the *callee-tree* can refine it.


*FTfJP '21, July 13, 2021, Virtual, Denmark*

ACM ISBN 978-1-4503-8543-5/21/07.
https://doi.org/10.1145/3464971.3468416

can distort program behaviour, resulting in an execution trace that does not reflect untraced program behaviour. Concurrency performance behaviour is especially fragile to such distortions given its natural non-determinism. If these distortions are too great, the resulting analysis could be invalid.

To enable the exploration of all execution paths in an APG, it must represent the entire program. As many components of a program will be pre-compiled `.jar` libraries (both the Java standard library and third party libraries), *Scopda* supports generating APGs from both Java source code and JVM bytecode. The APG is fundamentally language agnostic and a single APG may contain subgraphs generated from multiple formats. For each supported input format, *Scopda* defines a *language interface* which converts the format to APG subgraphs and, for some formats (i.e. source code but not bytecode), renders APG subgraphs back into the source format.

Section 4 explores language features beyond those used in the running example (e.g. conditional control flow, inheritance, and unstructured JVM bytecode). Most features are naturally handled by the semantic *lowering* involved in generating an APG. However, some unstructured patterns in JVM bytecode cannot be represented in structured formats such as the APG. *Scopda* handles such patterns at the per-function level using a *grey box* (a simplified representation) approach, whereby precision is sacrificed while preserving safety.

To validate *Scopda*'s approach in the real-world, we apply it to *improvement specifications* generated by *Rehype* for a large real-world Java program (c. 500kLoC). *Scopda* successfully generates (sensible) source-code patches for nineteen suggested *improvements* (Section 5).

The contributions of this paper are:

(1) A method for generating concrete source code *patches* based on *improvements* identified using dynamic analysis and *specified* with dynamic-domain data.
(2) A method for mapping dynamic traces to nodes in a static code graph.
(3) A new static program representation, the *abstract program graph*, that enables: dynamic-to-static mapping, reflecting edits made on the graph back into the original source code, and supporting multiple source formats (e.g. Java source code and JVM bytecode) within a single graph.

Finally, Section 6 positions this work among related work and Section 7 concludes the paper.

## 2 RUNNING EXAMPLE

We concentrate on the "convert-to-inline" optimisation – where *Rehype* suggests inlining a spawned task to improve a program's concurrency performance (an example is given in Fig. 1). The convert-to-inline optimisation can be applied to certain concurrent tasks to improve resource usage efficiency, see *Rehype* [7] for more details. *Rehype* specifies the task to be inlined using the *caller-path* for the task-spawn invocation and the *callee-tree* of the task execution. The *caller-path* and *callee-tree* will not be contiguous on a thread in most instances (i.e. the *callee-tree* will not be rooted at the end of the *caller-path*), as the callee-tree will be invoked from a separate thread. *Scopda* identifies the specific task-spawn *static location*, and derives additional *patch points*, and generates a source-code *patch* encapsulating the *improvement*. *Patch points* are source-code locations that are concomitant with the *static location* (e.g. usages of the `Future` variable the task-spawn function returns). If the task-spawn *static location* is ambiguous, *Scopda* can:

(1) generate multiple *patches*, one for each *static location* (the user can choose which to apply);
(2) report the ambiguity as an error; and/or
(3) generate a new *Rehype trace-config* that, when used for a new program execution, will capture *trace* data sufficient to disambiguate the *static location* in future uses of *Scopda*.

Fig. 1a shows, in JSON format, the *improvement specification* generated by *Rehype* (simplified[3] for readability), and Fig. 1b shows a source code *patch* to implement the *improvement*. In this example `calculateNumber()` contains the task-spawn call and `queryDatabase()` is the *task-body function*. The task framework consists of the spawning function `ExecutorService.submit()` and the root task execution function `Future.run()` (these are the standard task concurrency utilities in Java 8, see the `java.util.concurrent` package documentation [4]). The first invocation in the `calleeTree` is `Future.run()` as it is the first (*tracked*) function called by the thread that executes the task. It, in turn, calls `queryDatabase` via the callback (e.g. a Java `Runnable`) given when spawning the task. Given the *improvement specification*, *Scopda* identifies (Fig. 1b)

- *(static-location)* the `es.submit()` call at line 2 as the task-spawn call to modify;
- *(patch-point)* the *task-body function* `queryDatabase`; and
- *(patch-point)* the `databaseResult.get()` call at line 5 as a use of the task result that should also be modified (as it uses `Future.get()` to wait for the task result).

The modified source code calls the *task-body function* directly at line 2, updates the returned variable's type, and removes the `Future.get()` call, instead using the variable directly, at line 5.

Determining the benefit of the convert-to-inline optimisation (i.e. identifying it as an improvement) in this example requires dynamic analysis. It is unclear from a purely static perspective whether the elided code (line 3 of Fig. 1b) performs significant work or not. In the former case we should retain `queryDatabase` as a task, but in the latter we should inline the spawn as a direct call. By contrast, dynamic analysis can determine how much work the elided code performs and thus whether inlining `queryDatabase` is beneficial.

## 3 METHOD

*Scopda* contains three primary components (illustrated in Fig. 2): a *language interface*, *dynamic-static mapper*, and *change transformation functions*. The language interface (LI) generates APGs from source-code (and JVM bytecode) and renders APGs back into source-code. The dynamic-static mapper (DSM) takes a *dynamic context* and an APG and returns the corresponding set of *static locations* – in effect, nodes in the APG. A change transformation function (CTF) takes an APG and *static locations* as input and transforms the APG to implement the *improvement*. One CTF is implemented for each

[3] Specifically, method names are not qualified by type signature or containing class, and `interface` names such as `Future` are treated as if they were `class` names.

```
{
  "optimisationType": "convertToInline",
  "callerPath": ["main", "calculateNumber", "ExecutorService.submit"],
  "calleeTree": {
    "function": "Future.run",
    "children": [
      { "function": "queryDatabase", "children": [ ] }
    ]
  }
}
```

**(a) A JSON-format *improvement specification*.**

```
1: static int calculateNumber(ExecutorService es) {
2:- Future<Integer> databaseResult = es.submit(
        ThisClass::queryDatabase);
2:+ int databaseResult = queryDatabase();
3:  // ... elided code ...
4:  int otherValue = other();
5:- return databaseResult.get() + otherValue;
5:+ return databaseResult       + otherValue;
6: }
7:
8: static int queryDatabase() { /* ... */ }
9: static int other() { /* ... */ }
```

**(b) Source-code patch generated by *Scopda*.**

Rehype: "Execution tracing for the program above has suggested performance gains from: inlining the task that is spawned at the *dynamic context* corresponding to the `"callerPath"` that, when executed, invokes the `"calleeTree"`."
Scopda: "Suggested patch is: calling `queryDatabase` directly from `calculateNumber`, instead of spawning a task for it; concomitantly when calling `queryDatabase` directly, the result is an `int` and does not require a `.get()` call to get the value."

**Figure 1: Informal explanation of an example "convert-to-inline" *improvement* proposed by *Rehype* (a) and implemented by *Scopda* (b).**

*optimisation*. The appropriate CTF to use is determined by the *optimisation type* in the *improvement specification*. The *Scopda* process is thus:

**step 1**: *(LI)* generate the APG;
**step 2**: *(DSM)* map *dynamic contexts* to *static locations*;
**step 3**: *(CTF)* transform the APG for the *improvement*;
**step 4**: *(LI)* render the transformed APG back into source-code; and
**step 5**: generate a patch (a git-style diff) by comparing the rendered source-code to the original.

## 3.1 Abstract Program Graph

An APG is a unified graph structure containing a program's inter-procedural control flow and local data flow as well as structural (AST-like) information. It has 4 node types (function, variable, operation, and branch) and 15 edge types. It is edge-centric – most semantic details are encoded by the edges. A key property is that it can be rendered back to the unique input AST that generated it, and hence back into the original source-code (modulo white space and spurious bracketing).

The APG contains the semantic information of an AST, its call graph, and its control flow graph. While existing representations, such as graph-overlays [9], contain equivalent information and are equally effective for query and analysis, *Scopda* also needs to modify code for patch generation; the APG's unified design is better suited to transformation as it does not need to coordinate the transformation of multiple overlays.

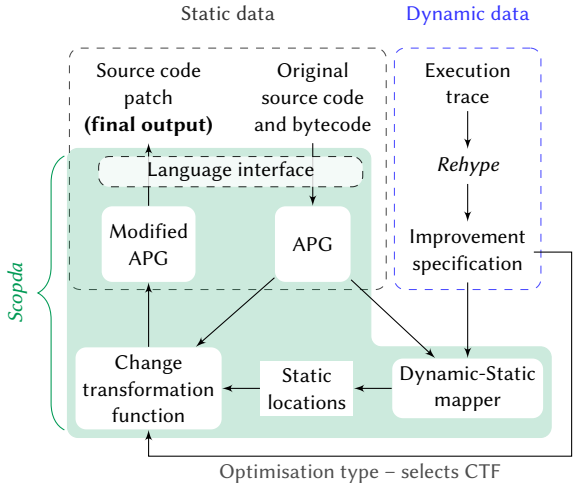


**Figure 2: High-level structure of *Scopda*. Arrows indicate information flow.**

Fig. 3 illustrates the APG for the `calculateNumber` method from Fig. 1 (pre- and post-change). The operation nodes (circles) resemble an AST statement and expression tree. Though the function implementations are elided, the `<calls>` edges connect directly to the target function nodes, providing the call-graph information. The operation and branch nodes along with the solid (structural) and dashed (control flow) edges define the control flow. The variable nodes and zigzag edges define the data flow.

The APG design prioritises simplicity of analysis and transformation over simplicity of APG-to-source-code rendering. This priority takes form as a guiding principle: maintain a narrow interface while retaining full expressiveness of (Java) programs. An example of the narrow interface is the simple control flow constructs: standard control flow is described with `branch` nodes and `<goto>` and `<returns>` edges, while exceptional control flow is represented via `<on-exception-goto>` edges (effectively a `catch` statement).

When generating an APG (*Scopda* **step 1**), a series of *lowering* transformations are applied to normalise source-code representation:

- Local variables are represented in Single Static Assignment (SSA) [10] form (SSA $\phi$-functions are represented as independent variable nodes with `<ssa-predecessor>` edges to their SSA *versions*).
- Unnamed variables use A-Normal Form [11].
- Non-local variable accesses become calls to *intrinsics*. An intrinsic is a function node representing a language-specific operation, such as accessing an object field or referencing a method. Intrinsics provide a unified approach to various language features and make shared memory access explicit, without affecting the semantics of the program.
- Operators are also converted into intrinsics (e.g. the plus operator in the running example becomes an `Intrinsic::Operator::Plus` function node in Fig. 3).
- All functions are represented statically (i.e. `'this'` parameters are explicit).

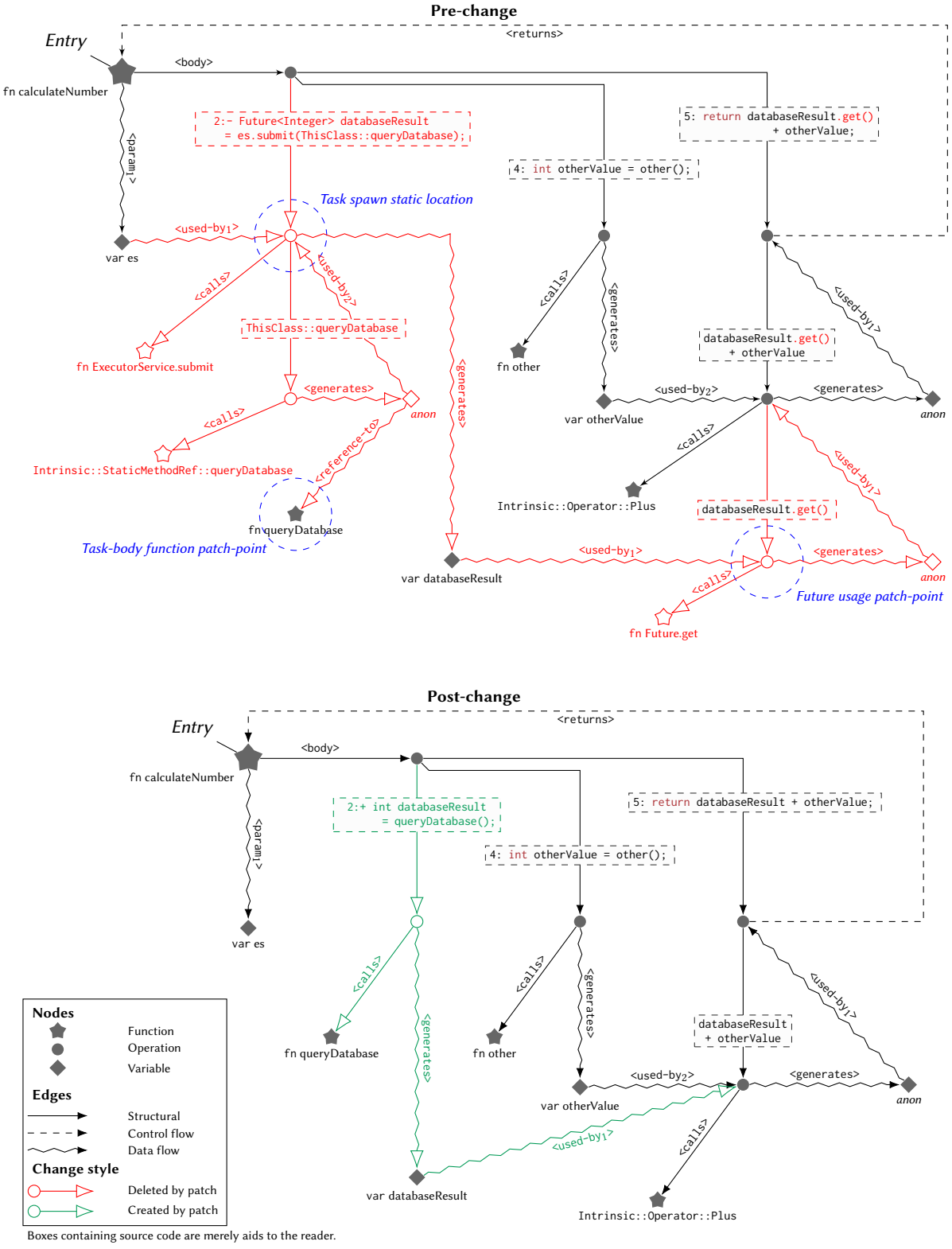


**Figure 3: APG representations of `calculateNumber` before and after the patch from Fig. 1b. Hollow nodes and edge arrowheads are those deleted or created by the patch (these also follow git-diff colour conventions).**

- Function and variable symbols are resolved (e.g. `<calls>` edges terminate at function nodes which are also the root of the function implementation graph).

While the APG defines a narrow interface and retains full semantic expressiveness, rendering back into the input AST may require language-specific information. LIs annotate nodes with this language-specific information during APG construction. For example, the Java LI annotates branch nodes to indicate the type of branch statement (e.g. `if`, `for`, `while`). These annotations are opaque to the rest of *Scopda*, but are associated with all copies of a node (CTFs may duplicate nodes).

## 3.2 Dynamic-Static Mapper

The dynamic-static mapper converts a given *dynamic context* into a set of *static locations* (*Scopda* **step 2**) by exploring static execution paths in the APG. To identify the *static locations* that correspond to a *dynamic context*, the mapper identifies all possible *static code paths* (SCPs) that can correspond to a *caller-path* (this method generalises to *callee-trees* as a tree can be treated as a series of paths). An SCP is a series of nodes in the APG connected by control flow edges. An SCP is *valid* for a *caller-path* if the SCP, filtered to *tracked* function nodes, is equal to the *caller-path*. Each node within an SCP is a *static location*. Once the SCPs are calculated, the CTF for the optimisation selects relevant *static location(s)* from the SCPs. For example, the key *static location* in our running example (Fig. 1) is the operation node for the call to `ExecutorService.submit` (highlighted in the topmost dashed (blue) circle in Fig. 3).

In simple cases, including Fig. 1, every *dynamic context* maps to a single *static location*. In richer situations, such as when a function can be called from two different *dynamic contexts*, a *dynamic context* may map to multiple *static locations*.

*Scopda*'s execution path exploration is intuitively a guided graph traversal along APG control flow paths. It identifies SCPs that are *valid* for a given *caller-path* by testing all control flow paths starting at the first invoked function in the *caller-path* and terminating at the last invocation in the *caller-path*. A control flow path is invalidated (and exploration on it is terminated early) when it encounters a *tracked* function node that does not correspond to the next invoked function in the *caller-path*.

A naive approach might generate a set of concrete SCPs using standard traversal, but this would not scale to real-world programs. Such an approach would result in a combinatorial explosion of SCPs when there are many possible static code paths between *tracked* functions (in fact, there may be infinite paths given recursion). Moreover, the number of possible paths makes it computationally infeasible to sequentially iterate over them.

Instead, when given a *caller-path*, *Scopda* uses the call graph to produce a *caller-path*-enhanced call graph. The paths in this *caller-path*-enhanced call graph are exactly the set of valid SCPs. In this graph, invocations, that are adjacent in the *caller-path*, are interposed by a graph representing the SCPs of non-*tracked* functions between the invoked functions.

More concretely, given a caller-path $p$ and an invocation $p_i$ within it, and writing $G$ for the program's call graph, define $G^{\text{forw}}(p_i)$ to be the subgraph of $G$ which can be reached from $p_i$ without passing through a *tracked* function node. Similarly define $G^{\text{back}}(p_{i+1})$ as the subgraph of $G$ which is backwards reachable from $p_{i+1}$ without passing through a *tracked* function node. Then, the graph of SCPs that interposes two adjacent invocations, $p_i$ and $p_{i+1}$, is calculated as $G^{\text{forw}}(p_i) \cap G^{\text{back}}(p_{i+1})$ – the non-*tracked* sub-call-graph that is reached from $p_i$ and can reach $p_{i+1}$.

## 3.3 Change Transformation Functions

*Scopda*'s CTFs implement *improvements* (*Scopda* **step 3**) by transforming the APG (e.g. by inlining a task's execution). CTFs take an APG and a set of *static locations* as input, and generate a transformed APG. CTFs are implemented as a series of graph analyses and transformations applied to the original APG.

For example, in the running example (Fig. 1), the *convert-to-inline* CTF's process is:

(1) Derive additional *patch points* (Section 2) from the task spawn *static location* (the `submit()` call) by identifying:
   (a) The second argument to the `submit()` call (the `<used-by$_2$>` edge terminating at the task spawn static location in Fig. 3) is the *task-body function*.
   (b) The `Future.get()` call, that takes the `submit()` call's result variable (`databaseResult`) as the first argument, must be removed.
(2) Modify the `submit()` call to call the *task-body function*.
(3) Make the result variable, `databaseResult`, an `int` instead of a `Future<Integer>`.
(4) Remove the call to `Future.get()` and replace it with a simple usage of the `databaseResult` variable.

This is illustrated in Fig. 3, where the top and bottom graphs are pre- and post-change versions, respectively. Edges changed between the versions (corresponding to the patch) have enlarged arrow heads and nodes created/deleted are hollow. Modified edges and nodes also follow git-diff colour conventions.

## 3.4 Rendering Source Code

There are two subtleties in the APG-to-source-code rendering process (*Scopda* **step 4**). First, APG to AST conversion is done per-function; higher-level constructs, such as classes, are rendered unchanged (save for the modified functions). Since *Scopda* operates at the function level, for both analysis and transformation, no other source code needs to be changed. Second, the final patch is minimised at each output step (APG-to-AST and AST-to-source-code); this is important for adoption as larger and more complex patches are harder for developers to check.

At each output step, *Scopda* copies as much of the original data (AST nodes or source code text) as possible and only generates new versions for the parts that have been modified. At the APG to AST step, *Scopda* copies the original AST for the function being rendered and replaces only those nodes that correspond to modified APG subgraphs. Similarly, at the AST-to-source code step, if an AST node is unmodified the corresponding source code is copied, whereas if it is modified the source code is generated from the AST. This process of using the original data where possible ensures that the resulting diff is minimal (and non-functional aspects of the code, such as code style and comments, are retained).

Finally, the output patch (*Scopda* **step 5**) is generated by applying git's diff algorithm to the original source code and generated source code.

## 4 REAL-WORLD COMPLEXITY

The running example does not include various features present in real-world programs, such as branch statements, inheritance, unavailable source code, and unanticipated binary code structures. We now summarise how the APG, and *Scopda* more generally, handles these, to illustrate *Scopda*'s approach to real-world complexity.

**Branching** While branching statements can generate various branch orders and behaviours (e.g. `if` vs. `switch` statements), such distinctions are irrelevant to the analysis and transformation performed by *Scopda*. The dynamic-static mapper explores all branch options irrespective of order. Similarly, CTFs that modify branching statements consider the branches as identified by the *static locations*, not their order. Therefore, the APG identifies branches as distinct from each other, but does not specify *how* they are distinguished by the source language (LI annotations can indicate this for source-code rendering).

**Inheritance and interfaces** Though omitted in the running example, the APG supports inheritance with an `<overrides>` edge linking a function implementation to an interface function it inherits/overrides. In analysis these edges are, effectively, expanded as `<calls>` edges between operation nodes that call the interface function node and the implementation function nodes. *Scopda* does not (yet) perform call-graph reduction using infeasible path analysis [13] as the dynamic-static mapper's reachability-based approach for constructing graphs of SCPs is currently sufficient for the analyses it performs.

**Source-code availability** APGs must contain all *tracked* functions and all functions on static call paths between *tracked* functions to enable dynamic-to-static mapping. In many cases this includes functions for which the source code is unavailable (e.g. third-party libraries). As such, *Scopda* also implements a *language interface* for generating APGs from JVM bytecode, though it cannot render bytecode derived nodes into Java source code. Individual APGs may contain subgraphs derived from multiple input formats/languages (e.g. JVM bytecode and Java source code).

**JVM bytecode and grey boxes** JVM bytecode is an unstructured format in which there are valid bytecode patterns[4] that cannot be represented in structured graphs (Miecznikowski and Hendren [6]), such as an APG. These patterns are addressed by an additional APG construct, the *grey box*. Grey boxes contain only those nodes and edges to enable inter-procedural analysis (e.g. call-graph edges) by the dynamic-static mapper; they do not support code modification. Grey boxes provide safe, but imprecise, information. Moreover, grey boxes permit *Scopda* to follow a *fall back to simpler representation* principle when encountering unanticipated structures in code; this enables support for unstructured formats and eases support for new languages. While grey boxes have the potential to reduce analysis precision, they should not impede final patch generation (and do not in our experimental experience (Section 5)) as they never represent source-available functions (i.e. code which patches affect).

```
// Track main(), g(), and h(), but not untracked1() nor untracked2().
void main() {
  if (cond) {
    untracked1();
  } else {
    untracked2();
  }
}
void untracked1() { g(); h(); }
void untracked2() { h(); g(); }
```

**Figure 4: Representing `untracked1` and `untracked2` as *grey boxes* causes ambiguity in *Scopda*'s analysis.**

As an example of the analysis imprecision that grey boxes can cause, Fig. 4 presents a constructed case that, with grey boxes, could cause ambiguity in *Scopda*'s analysis. In the example, functions `main`, `g`, and `h` are tracked and functions `untracked1` and `untracked2` are not. The `main` function calls `untracked1` and `untracked2`, which both call `g` and `h` but in different orders. Given a trace-log showing `main` calling `g` and then `h` in turn, *Scopda* could determine that `untracked1` was called (due to the order of the `g` and `h` invocations). However, if `untracked1` and `untracked2` were both grey boxes, *Scopda* would not be able to determine which one was called as both would indicate that they call `g` and `h`, but would not indicate the order. If a patch then depended on which was called (e.g. a patch that needs to edit `main` in some way), this ambiguity could impede patch generation.

**Implementation** In practice it is faster to generate APGs from JVM bytecode instead of raw `.java` files as it does not require further compilation (raw `.java` files must be compiled by `javac` to resolve types for the APG, whereas these are already resolved in bytecode). As the APG representations of bytecode and source-code versions of a function are semantically equivalent, we initially generate the entire program APG from bytecode and only generate APG function representations from source-code when necessary. Specifically, when source-code is available for a function *Scopda* uses it if

- the bytecode version cannot be successfully generated (i.e. it would be a grey box); or
- the CTF must transform the function to generate a patch.

If the CTF attempts to transform a source-unavailable function, an error is returned.

While this approach requires generating APG subgraphs twice for functions to be edited (first from bytecode and then from source code), potentially presenting scalability issues for large code bases, in practice this is not a significant issue. APG generation is constant w.r.t. the number of `.java` files to be edited for a given set of patches. Only those `.java` files that will be edited in at least one patch are converted into an APG, the rest of the source code can be ignored (the vast majority of code in large projects). Ignoring non-patch-relevant code during (`javac`) compilation is possible as the compiled `.jar` file – which is executed to generate the original trace and is also used as the basis of initial APG construction – can be given as a

[4]Note that these unrepresentable patterns are **structural** patterns, such as multi-entry basic blocks, not **behavioural** patterns that enable dynamic behaviour, such as Android Intents. Behavioural patterns can be represented in the APG format just as any in any other structured format (e.g. Java). Fundamentally, any structural pattern that can be represented in a structured programming language can be represented as an APG.

```
public ~~UncheckedFuture<~~ByteBuffer~~>~~ sign(ByteBuffer value, Key key) {
  ~~return executorService.submit(() -> {~~
    try {
      SecretKey secretKey = getKey(key, true);
      Mac mac = Mac.getInstance(secretKey.getAlgorithm());
      mac.init(secretKey);
      return ByteBuffer.wrap(mac.doFinal(value.array()));
    } catch (NoSuchAlgorithmException | InvalidKeyException e) {
      throw new RuntimeException(e);
    }
  ~~});~~
}
```

**Figure 5: An example patch generated by *Scopda* for a real-world Java server.**

class-path library to the `javac` compiler, providing all dependencies required by the `.java` files. Naturally, if a `.java` file is affected by multiple patches, it only needs to be compiled once as the same APG can be used to generate multiple patches.

## 5 APPLICATION TO REAL-WORLD

We evaluate *Scopda* on a (proprietary) industrial Java API server (c. 500kLoC) for a consumer web and mobile application – the same software *Rehype* was evaluated against. *Scopda* successfully generates source-code patches for each of the nineteen *improvements* suggested by *Rehype* (discussed in the results section of [7]). Fig. 5 contains one of the real patches generated by *Scopda*.

The APG is constructed from the server's source code `.java` files, compiled `.jar` file, and the Java 8 SE standard library `.jar` file. The final APG contains 8 690 403 total nodes, 409 231 functions, and 223 492 intrinsics. Of these, the JVM bytecode *language interface* successfully generates APG representations for 398 699 functions, 97.43% of all functions. *Grey box* versions are generated for the remaining functions (all are in third-party `.jar` files where source is unavailable).

The server with patches automatically applied, by `git apply`, successfully executes and exhibits the predicted performance increase (see *Rehype* [7]).

## 6 RELATED WORK

We are unaware of existing work that transforms static source code based on dynamic analysis.

Work relevant to the APG includes established general representations such as control flow graphs [1] and call graphs, system-specific internal representations (e.g. LLVM's IR [5]), and task-specific representations such as source-code query graphs [8, 9], among others [12]. These differ from the APG in specificity (e.g. CFGs), target use (query graphs), or structure (compiler IRs designed for optimisation and machine-code generation). By nature of the target application, the APG does not need to represent some information that is critical to other IRs, such as an efficiently checkable type-system, as it can assume the input programs are valid. Within the context of compiler IRs, the APG uses a form of semantic lowering that can be uniquely reversed (i.e. the lowered IR can generate the unique input high-level source code (modulo white space and spurious bracketing)), whereas most IRs do not require this attribute [3]. Furthermore, given its unified approach (as opposed to, for example, overlay graphs), the APG is particularly well suited to algorithms that combine analysis and transformation.

Some existing work combines static analysis and dynamic analysis for the detection of bugs [2, 14], especially multi-threading bugs which are particularly difficult to find with static analysis alone. Such work combines static analysis and dynamic analysis to enhance the analysis (detection) process, whereas *Scopda* combines them to translate dynamic analysis identified *improvements* back into the static domain (source code).

## 7 CONCLUSION

We introduced and evaluated *Scopda*, a system for generating source-code *patches* for *improvements* identified by execution-trace-based dynamic analysis, specifically the *Rehype* companion tool. *Scopda* first maps the dynamic-domain *improvement specifications* into the static domain using a general method and then applies a code transformation method specialised to each optimisation. *Scopda* uses a custom static program representation (abstract program graph, APG) to map between dynamic and static domains and perform code transformation within a single representation. We evaluated *Scopda* on a large real-world program and demonstrated that it generates sensible source-code patches.

In the future we plan to develop support for more optimisations and expand *Rehype* and *Scopda* to work on more programming languages. We also plan to incorporate further static analysis techniques, such as infeasible path analysis, to improve *Scopda*'s treatment of trace sparsity in *improvement specifications*.

## ACKNOWLEDGMENTS

We thank the referees for their helpful suggestions. The first author was funded by the Engineering and Physical Sciences Research Council (EPSRC), the Cambridge Trusts, and the University of Cambridge Department of Computer Science and Technology.

## REFERENCES

[1] Frances E. Allen. 1970. Control Flow Analysis. *SIGPLAN Not.* 5, 7 (July 1970), 1–19. https://doi.org/10.1145/390013.808479

[2] Cyrille Artho. 2005. *Combining static and dynamic analysis to find multi-threading faults beyond data races.* Ph.D. Dissertation. ETH Zurich. https://doi.org/10.3929/ethz-a-005005473 Diss., Technische Wissenschaften, Eidgenössische Technische Hochschule ETH Zürich, Nr. 16020, 2005.

[3] Fred Chow. 2013. Intermediate representation. *Commun. ACM* 56, 12 (2013), 57–62. https://doi.org/10.1145/2534706.2534720

[4] Java Documentation 2021. Java 8 `java.util.concurrent` documentation. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html. Accessed May 2021.

[5] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[6] Jerome Miecznikowski and Laurie Hendren. 2002. Decompiling Java bytecode: Problems, traps and pitfalls. In *International Conference on Compiler Construction.* Springer, 111–127. https://doi.org/10.1007/3-540-45937-5_10

[7] Indigo Orton and Alan Mycroft. 2021. Refactoring traces to identify concurrency improvements. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '21).* Association for Computing Machinery, 8. https://doi.org/10.1145/3464971.3468420

[8] Dileep Kumar Pattipati, Rupesh Nasre, and Sreenivasa Kumar Puligundla. 2020. OPAL: An extensible framework for ontology-based program analysis. *Software: Practice and Experience* (2020). https://doi.org/10.1002/spe.2821

[9] O. Rodriguez-Prieto, A. Mycroft, and F. Ortin. 2020. An Efficient and Scalable Platform for Java Source Code Analysis Using Overlaid Graph Representations. *IEEE Access* 8 (2020), 72239–72260. https://doi.org/10.1109/ACCESS.2020.2987631

[10] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. Association for Computing Machinery, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

[11] Amr Sabry and Matthias Felleisen. 1992. Reasoning about Programs in Continuation-Passing Style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming (LFP '92)*. Association for Computing Machinery, New York, NY, USA, 288–298. https://doi.org/10.1145/141471.141563

[12] James Stanier and Des Watson. 2013. Intermediate Representations in Imperative Compilers: A Survey. *ACM Comput. Surv.* 45, 3, Article 26 (July 2013), 27 pages. https://doi.org/10.1145/2480741.2480743

[13] Frank Tip. 2015. Infeasible paths in object-oriented programs. *Science of Computer Programming* 97 (2015), 91–97. https://doi.org/10.1016/j.scico.2013.11.005

[14] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: Detecting Concurrency Bugs through Sequential Errors. *SIGARCH Comput. Archit. News* 39, 1 (March 2011), 251–264. https://doi.org/10.1145/1961295.1950395